Dynamo: Amazon's Highly Available Key-value Store

Author: Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Presentation: Shijie Xu, Ying Wang
Why Dynamo?

- Fully Managed
- Fast, Consistent Performance
- Highly Scalable
- Flexible
System Assumptions and Requirements

- Query Model: simple read and write operation to a small data item that is uniquely identified by a key

- ACID Properties: Atomicity, (Weaker) Consistency, (No) Isolation, Durability

- Efficiency: Latency requirements which are in general measured at the 99.9th percentile of the distribution

- Other Assumption: Only deal with benign failures
Service Level Agreements

- Application can deliver its functionality in a bounded time

Fig-1 Service-oriented architecture of Amazon's platform
Design Consideration

● Sacrifice strong consistency for availability
● Always writeable
● Conflict resolution is executed during read instead of write
● Other principles:
  ○ Incremental scalability
  ○ Symmetry
  ○ Decentralization
  ○ Heterogeneity
# System Architecture

## Core techniques used:
- Partitioning
- Replication
- Versioning
- Membership
- Failure handling

### Table 1: Summary of techniques used in Dynamo and their advantages.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available.</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees</td>
<td>Synchronizes divergent replicas in the background.</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection</td>
<td>Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.</td>
</tr>
</tbody>
</table>
API

- **get(key)**
  Returns
  A single object or a list of objects, and
  A context

- **put(key, context, object)**
  Uses *key* to determine the write replicas
  Writes the replicas to disk

- **Context**
  System metadata about the object
Partitioning

- Consistent Hashing
  - The output range of hashed values treat as a “ring”
  - Pros: incrementally scalable, adding a single node does not affect the system significantly
  - Cons: lead to the uneven distributed load, and oblivious to the heterogeneity in the performance of nodes.
Partitioning

- “Virtual Node”
  - Each node can be responsible for more than one virtual node
  - Work distribution proportional to the capabilities of the individual node
Replication

- Each data item is replicated at N hosts
- Preference list: The list of nodes that is responsible for storing a particular key
  - May contain more than N nodes due to failures
  - Contains only distinct physical nodes
Replication

- **Example: N = 3**
  - Node B replicates the key k at node C and D in addition to storing it locally
  - Node D will store the keys in the range (A,B], (B,C], and (C,D]

![Diagram of key replication in Dynamo ring](image)

Fig-4 Partitioning a replication of keys in Dynamo ring.
Data Versioning

- System is eventually consistent, thus a `get()` call may return many versions of the same object

- Challenge: An object can have distinct version sub-histories, the system needs to reconcile in the future

- Solution: Syntactic reconciliation and Semantic reconciliation
Vector Clock

- A vector clock is a list of [node, counter] pairs
- Versioned object -> vector clock
- Update an object, put(key, context, object)
- “context” is obtained from an earlier read operation, which contains the vector clock information
Syntactic reconciliation
What if?

Source: Rick and Morty S02E01
Semantic reconciliation

- Failures + concurrent updating => version branching
- Collapse
- Version branching is resolved by data store or the application
  - Data store: latest write wins
  - Application: merge
Vector Clock Issue

● Vector clock may grow when many servers coordinate the writes to one object
● Truncation Scheme
  ○ Delete the oldest [node, counter] pair when the number of pairs reaches a threshold
● More issue: Inefficiencies in reconciliation because of missing information
  ○ Not show in production
Client request choices

- Generic load balancer
  - No code specific to Dynamo
  - Extra request forwarding step
- Partition-aware client library
  - Better performance
Execute get and put: Quorum in Dynamo

- The first reachable node in the preference list is the coordinator
- R: minimum number of nodes that must participate in successful read operation
- W: minimum number of nodes that must participate in a successful write operation
- Setting R+W > N yields a quorum-like system
- The latency of a get() (or put()) operation is dictated by the slowest of the R (or W) replicas
- R and W are usually configured to be less than N, to provide better latency
Execution of *get()* operation

*get()*
- Coordinator requests reading from N nodes, waits for R responses
- If the responses agree, returns the object with context
- If they disagree
  - If they are causally related, returns the most recent value
  - If they are causally unrelated, returns all versions
Execution of `put()` operation

`put()`:
- Coordinator generates new version vector clock and writes new version locally
- Forwards metadata to highest ranked reachable nodes in the preference list
- Waits for W-1 or more writes to be succeed
Handling Failures: Hinted Handoff

- “Always writeable”
- Avoid the read and write operations failure, due to temporary node or network failures

Fig-6 Partitioning a replication of keys in Dynamo ring.
Handling permanent Failures: Replica Synchronization

● Merkle tree:
  ○ Parent node are hashes of (immediate) children
  ○ Comparison of parents at the same level tells the difference in children
  ○ Does not require transferring entire (key, value) pairs
The power of gossip
The power of gossip

- Ring Membership
  - All nodes exchange their membership histories
  - Each node randomly contact a peer every second
  - Eventually consistent
  - Each node forward a key’s read/write operations right set of nodes directly
The power of gossip

● External Discovery
  ○ Nodes may not know each other - logical partitions
  ○ Seed Nodes to avoid logical partitions
The power of gossip

● Failure Detection
  ○ Detect failure locally is sufficient
  ○ Periodically retry failed node(s)
  ○ No need for a decentralized failure detector
Implementation

- Java
- Local persistence component allows for different storage engines to be plugged in:
  - Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes
  - MySQL: object of > tens of kilobytes
Main modes of Dynamo

● Business logic specific reconciliation
  ○ Merge
  ○ Application-specific reconciliation

● Timestamp based reconciliation
  ○ Last write wins

● High performance read engine
  ○ $R = 1, W = N$, Dynamo provides the ability to partition and replicate their data across multiple nodes thereby offering incremental scalability
Experiences

- **N**: durability
- **W and R**: availability, durability and consistency
  - Increase **W** can increase durability but reduce availability
- 
  \[ (N,R,W) = (3,2,2) \] provides satisfying performance, durability, consistency, and availability SLAs
Performance

- Guarantee Service Level Agreements (SLA)
  - Latencies: diurnal pattern (incoming request rate)
  - Write latencies >> Read latencies

- Latencies around 200ms

Fig-9 Average and 99.9 percentiles of latencies for read and write request during peak season of Dec.2006
Better Performance

- Trade durability for better performance
- Each storage node maintains an object buffer in its main memory
- Write objects in buffer to disk using a writer thread periodically
- Read from buffer in memory

Fig-10 Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours
Balance

- Out of balance
  - If the node’s request load deviates from the average load by a value more than a certain threshold (here is 15%)
- Imbalance ratio decreases with increasing load
- Under high loads, a large number of popular keys are accessed and the load is evenly distributed

Fig-11 Fraction of nodes that are out of balance, and there corresponding request load. The interval between ticks in x-axis correspond to a time period of 30 mins.
Partitioning and placement of key

Strategy 1: T random tokens per node and partition by token value
- Slow bootstrapping process
- Recalculation of the Merkle tree
- Data partitioning and data placement are intertwined

Strategy 2 and Strategy 3
- Equal size partitioning strategies to distribute load uniformly
Server-driven and Client-driven Coordination

- Use a state machine to handle incoming requests
- Move the state machine to the client nodes
Balancing background & foreground

- Each node performs both background and foreground operation
- Background trigger resource contention
- Admission controller: change the runtime slices of the resource for background
Conclusion

● Dynamo is a highly available and scalable data store for Amazon’s e-commerce platform.

● Techniques:
  ○ **Gossiping** for membership and failure detection
  ○ **Consistent hashing** for node and key distribution
  ○ **Object versioning** for eventually – consistent data objects
  ○ **Quorums** for partition/failure tolerance
  ○ **Merkle tree** for resynchronization after failures/ partitions
Questions?
Handling permanent Failures: Replica Synchronization

- Comparing two nodes that are synchronized
  - Two (key, value) pairs: \((k_0, v_0)\) & \((k_1, v_1)\)

![Fig-7 Replica Synchronization](image)
Handling permanent Failures: Replica Synchronization

- Comparing two nodes that are **not synchronized**
  - Two (key, value) pairs: \((k_0, v_0)\) & \((k_1, v_1)\)

![Fig-8 Replica Not Synchronization](image)
Partitioning and placement of key

Strategy 1:
- T random tokens per node and partition by token value

Problems:
- Slow bootstrapping process
- Recalculation of the Merkle tree
- Complicated archival process

Fig-12 Partitioning and placement of key, strategy 1
Strategy 2:

- T random tokens per node and equal sized partitions
- Divides the hash space into Q equally sized partitions
Partitioning and placement of key

Strategy 3:
Q/S tokens per node, equal-sized partitions

- Divides the hash space into Q equally sized partitions
- Each node is assigned Q/S tokens
Thank you